AMPL

Introduction and manual

Robert Fourer

Northwestern University

David M. Gay

AMPL Optimization LLC

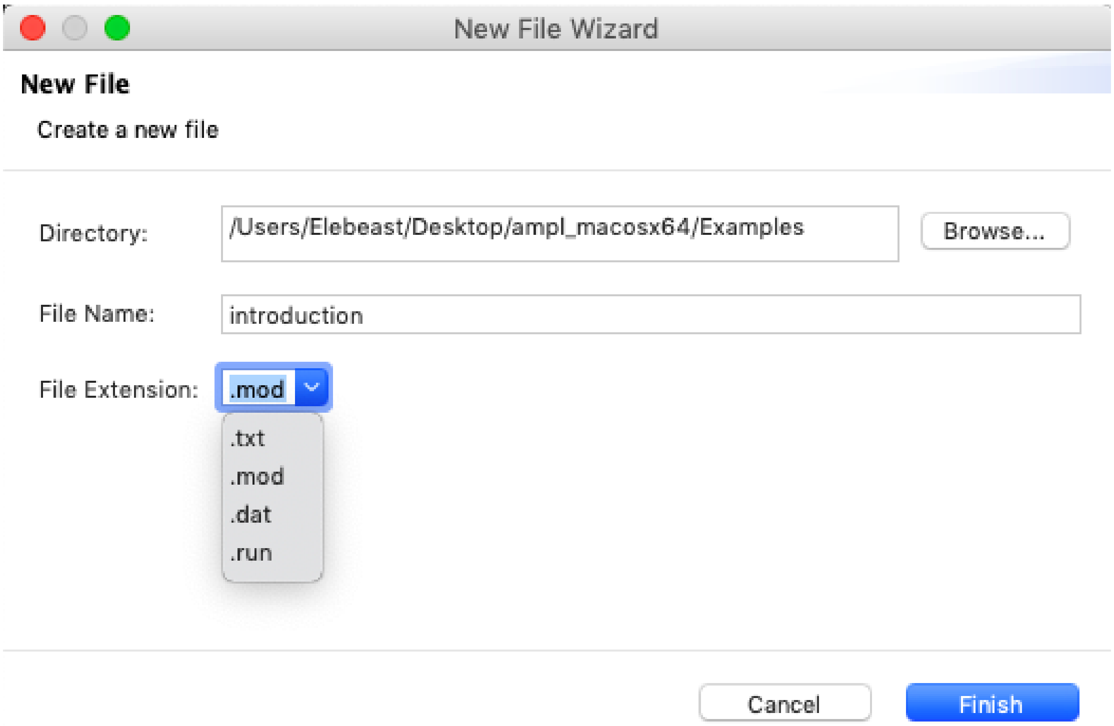
Brian W. Kernighan

Princeton University

CONTENT

INTRODUCTION

DOCUMENT TYPES



THE BASIC FILES ARE:
.MOD - USED TO DECLARE THE ELEMENTS OF THE MODELS: VARIABLES, OBJECTIVE, CONSTRAINTS AND DATA (SETS AND PARAMETERS).
.DAT - USED TO DEFINE THE DATA FOR THE MODEL.
.RUN - WHERE VARIABLE CONFIGURATIONS ARE DEFINED, “SCRIPTING CONSTRUCTS,” SUCH AS READING TABLES OR DATA BASES.

SINTAXE:

1. VARIABLE: VAR *VARIABLENAME*;
2. OBJECTIVE: MINIMIZE OR MAXIMIZE *OBJECTIVENAME*: . . . ;
3. CONSTRAINT: SUBJECT TO *RESTRICTIONNAME: . . .* ;

REMARKS:

1. EVERY LINE INSTRUCTION MUST BE TERMINATED WITH ";".
2. LINE COMMENTS ARE PRECEDED BY THE SYMBOL "#".
3. BLOCK COMMANDS ARE ENCLOSED BY THE SYMBOLS "//*. . . *//".
4. AMPL IS "CASE-SENSITIVE". I VARIABLE NAMES MUST BE UNIQUE.

→AMPL is similar to mathematical approach, see example below.

EXAMPLE

Pen Deals produces two colors of pen, blue and black.
Blue pen is sold for 1euro per pen, while black pen is sold for 1.5 euro per pen.
The company owns a process plant which can produce one color at a time.
However, blue pen is produced at a rate of 40 pens per hour, while the production rate for black pen is 30 pens per hour.
Besides, the marketing department estimates that at most 860 pens of black color and 1000 pens of blue color can be sold in the market.
During a week, the plant can operate for 40 hours and the pens can be stored for the following week.
Determine how many pens of each pen should be produced to maximize week revenue.

Mathematical approach

1. Objective function: 1* # of sold blue pen + 1.5 * # of sold black pen
2. Constraints:
   a. 1/40* # of blue pen + 1/30* # of black pen
   b. blue pen <= 1000
   c. black pen <= 860
   d. blue pen, black pen >= 0

→Answer will be carried out

AMPL approach

(file name: example1)
(file extension: .mod)

```
var BluePen;
var BlackPen;
maximize Revenue: 10*BluePen + 15*BlackPen;
subject to Time: (1/40)*BluePen + (1/30)*BlackPen <= 40;
subject to BlueLimit: 0 <= BluePen <= 1000;
subject to BlackLimit: 0 <= BlackPen <= 860;
```

→ Then, input commands in console to carry out the answer

Console:

```
reset;
model example1.mod;
solve;
display BluePen, BlackPen;

display Revenue;

expand Time;
```

→ All in all, the required questions are model in .mod. There are also .dat and .run as well as different commands will be discussed later in this manual.

Chapter 1

This chapter include some fundamental maximization and minimization problems with different commands.

- **MAXIMIZING**

2 variable linear programme in AMPL

An (extremely simplified) steel company must decide how to allocate next week's time on a rolling mill. The mill takes unfinished slabs of steel as input, and can produce either of two semi-finished products, which we will call bands and coils. (The terminology is not entirely standard; see the bibliography at the end of the chapter for some accounts of realistic LP applications in steelmaking.) The mill's two products come off the rolling line at different rates:

Tons per hour:	Bands	200
	Coils	140

and they also have different profitabilities:

Profit per ton:	Bands	\$25
	Coils	\$30

To further complicate matters, the following weekly production amounts are the most that can be justified in light of the currently booked orders:

Maximum tons:	Bands	6,000
	Coils	4,000

The question facing the company is as follows: If 40 hours of production time are available this week, how many tons of bands and how many tons of coils should be produced to bring in the greatest total profit?

Mathematical approach:

Decision variables : XB, XC
Objective function: 25*XB + 30*XC =! Maximization
Constraints:
1. (1/200)*XB + (1/140)*XC <= 40
2. XB <= 6000;
3. XC <= 4000;

Non-zero variables: XB, XC >= 0

AMPL approach: (.mod)

```
var XB;
var XC;
maximize Profit: 25 * XB + 30 * XC;
subject to Time: (1/200) * XB + (1/140) * XC <= 40;
subject to B_limit: 0 <= XB <= 6000;
subject to C_limit: 0 <= XC <= 4000;
```

After constructing the description of the linear program, type few AMPL commands to show the results

(Boldedlines)

```
ampl: model prod0.mod;
ampl: solve;
MINOS 5.5: optimal solution found.
2 iterations, objective 192000
ampl: display XB, XC;
XB = 6000
XC = 1400
ampl: quit;
```

Remarks: The result can be showed by typing (display variables;) , however the step has to be done after typing (model filename;) and (solve;)

- **MINIMIZING**

```
set NUTR;
set FOOD;

param cost {FOOD} > 0;
param f_min {FOOD} >= 0;
param f_max {j in FOOD} >= f_min[j];

param n_min {NUTR} >= 0;
param n_max {i in NUTR} >= n_min[i];

param amt {NUTR,FOOD} >= 0;

var Buy {j in FOOD} >= f_min[j], <= f_max[j];

minimize Total_Cost:  sum {j in FOOD} cost[j] * Buy[j];

subject to Diet {i in NUTR}:
   n_min[i] <= sum {j in FOOD} amt[i,j] * Buy[j] <= n_max[i];
```

NUTR→ nutrients; FOOD→ food

For the command param, e.g. param cost {FOOD} >0, indicates the numerical value which means the

cost of food should be positive. Form the above picture, there are upper and lower limit of the food. As we have to make sure the maximum intake of food or nutrients has to be equal or greater than the minimum, we also need to put

`param f_max {j in FOOD} >= f_min[j];`   →food

```
param n_max {i in NUTR} >= n_min[i];    →nutrients
```

In reality, the real problem are more complicated, in most of the cases we have more than two variables. In the following, .dat will be shown.

```
set NUTR := A B1 B2 C ;
set FOOD := BEEF CHK FISH HAM MCH MTL SPG TUR ;

param:    cost   f_min   f_max :=
  BEEF    3.19     0      100
  CHK     2.59     0      100
  FISH    2.29     0      100
  HAM     2.89     0      100
  MCH     1.89     0      100
  MTL     1.99     0      100
  SPG     1.99     0      100
  TUR     2.49     0      100 ;

param:    n_min  n_max :=
   A       700   10000
   C       700   10000
   B1      700   10000
   B2      700   10000 ;

param amt (tr):
            A     C    B1    B2  :=
   BEEF    60    20    10    15
   CHK      8     0    20    20
   FISH     8    10    15    10
   HAM     40    40    35    10
   MCH     15    35    15    15
   MTL     70    30    15    15
   SPG     25    50    25    15
   TUR     60    20    15    10 ;
```

By using .dat, we can input the desired date in a matrix. In the above picture, it shows the variable cost is a positive and should not exceed 100 dollar. And the limited amount of vitamins intake are within 700-10000. Also the amount of nutrients regarding different types of food.

→ With the commands have been discussed above, solution will be carried out. (Use display to show the desired data)

Chapter 2

This chapter continues the previous chapter with more comprehensive transportation problem.

The higher degree of the variable the complicated it will be of the problem. Speaking of reality, we can refer to the number of suppliers and destinations. For example, how can we optimize the deliver cost with 3 suppliers to 7 destinations? AMPL approach will be shown in the following.

```
set ORIG;   # origins
set DEST;   # destinations

param supply {ORIG} >= 0;   # amounts available at origins
param demand {DEST} >= 0;   # amounts required at destinations

   check: sum {i in ORIG} supply[i] = sum {j in DEST} demand[j];

param cost {ORIG,DEST} >= 0;   # shipment costs per unit
var Trans {ORIG,DEST} >= 0;    # units to be shipped

minimize Total_Cost:
   sum {i in ORIG, j in DEST} cost[i,j] * Trans[i,j];

subject to Supply {i in ORIG}:
   sum {j in DEST} Trans[i,j] = supply[i];

subject to Demand {j in DEST}:
   sum {i in ORIG} Trans[i,j] = demand[j];
```

file name: transp.mod

The *check* statement indicate that the sum of the supplies has to equal to the sum of the demand.

After we formulate the approach, we can input the date to .dat

```
param: ORIG:  supply :=  # defines set "ORIG" and param "supply"
        GARY   1400
        CLEV   2600
        PITT   2900 ;

param: DEST:  demand :=  # defines "DEST" and "demand"
        FRA     900
        DET    1200
        LAN     600
        WIN     400
        STL    1700
        FRE    1100
        LAF    1000 ;

param cost:
         FRA  DET  LAN  WIN  STL  FRE  LAF :=
   GARY   39   14   11   14   16   82    8
   CLEV   27    9   12    9   26   95   17
   PITT   24   14   17   13   28   99   20 ;
```

file name: transp.dat

→ Then we can simply get the answer by typing commands in console:

model transp.mod;
data transp.dat;
solve;

Now, we will focus on building a lager models, simply means a model contains multiple data.

We discussed the model with supplier and destination above, and need less to say, products variables should also be considered in real life.

```
set ORIG;   # origins
set DEST;   # destinations
set PROD;   # products

param supply {ORIG,PROD} >= 0;  # amounts available at origins
param demand {DEST,PROD} >= 0;  # amounts required at destinations

   check {p in PROD}:
      sum {i in ORIG} supply[i,p] = sum {j in DEST} demand[j,p];

param limit {ORIG,DEST} >= 0;

param cost {ORIG,DEST,PROD} >= 0;  # shipment costs per unit
var Trans {ORIG,DEST,PROD} >= 0;   # units to be shipped

minimize Total_Cost:
   sum {i in ORIG, j in DEST, p in PROD}
      cost[i,j,p] * Trans[i,j,p];

subject to Supply {i in ORIG, p in PROD}:
   sum {j in DEST} Trans[i,j,p] = supply[i,p];

subject to Demand {j in DEST, p in PROD}:
   sum {i in ORIG} Trans[i,j,p] = demand[j,p];

subject to Multi {i in ORIG, j in DEST}:
   sum {p in PROD} Trans[i,j,p] <= limit[i,j];
```

There is one variable added in this example which is *Trans*. Now, not only we have to consider the cost but also the required amount of units. The total cost will be the transport unit*cost, however, the transport unit also has to be equal to the supple and demand. In the following picture, how the date is allocated will be shown.

```
set ORIG := GARY CLEV PITT ;
set DEST := FRA DET LAN WIN STL FRE LAF ;
set PROD := bands coils plate ;

param supply (tr):  GARY   CLEV   PITT :=
            bands    400    700    800
            coils    800   1600   1800
            plate    200    300    300 ;

param demand (tr):
           FRA   DET   LAN   WIN   STL   FRE   LAF :=
   bands   300   300   100    75   650   225   250
   coils   500   750   400   250   950   850   500
   plate   100   100     0    50   200   100   250 ;

param limit default 625 ;

param cost :=

 [*,*,bands]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
        GARY    30   10    8   10   11   71    6
        CLEV    22    7   10    7   21   82   13
        PITT    19   11   12   10   25   83   15

 [*,*,coils]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
        GARY    39   14   11   14   16   82    8
        CLEV    27    9   12    9   26   95   17
        PITT    24   14   17   13   28   99   20

 [*,*,plate]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
        GARY    41   15   12   16   17   86    8
        CLEV    29    9   13    9   28   99   18
        PITT    26   14   17   13   31  104   20 ;
```

First, we indicate the number of units supply and demand regarding to different cities. And, we can consider the cost regarding to the products from supplier to destination.

Speaking of production, period of time is also one of the limitation. In AMPL, *1…T* represent the set of integer 1 through T.

```
set PROD;      # products
param T > 0;   # number of weeks

param rate {PROD} > 0;           # tons per hour produced
param inv0 {PROD} >= 0;          # initial inventory
param avail {1..T} >= 0;         # hours available in week
param market {PROD,1..T} >= 0;   # limit on tons sold in week

param prodcost {PROD} >= 0;      # cost per ton produced
param invcost {PROD} >= 0;       # carrying cost/ton of inventory
param revenue {PROD,1..T} >= 0;  # revenue per ton sold

var Make {PROD,1..T} >= 0;       # tons produced
var Inv {PROD,0..T} >= 0;        # tons inventoried
var Sell {p in PROD, t in 1..T} >= 0, <= market[p,t]; # tons sold

maximize Total_Profit:
   sum {p in PROD, t in 1..T} (revenue[p,t]*Sell[p,t] -
      prodcost[p]*Make[p,t] - invcost[p]*Inv[p,t]);
               # Total revenue less costs in all weeks

subject to Time {t in 1..T}:
   sum {p in PROD} (1/rate[p]) * Make[p,t] <= avail[t];

               # Total of hours used by all products
               # may not exceed hours available, in each week

subject to Init_Inv {p in PROD}:  Inv[p,0] = inv0[p];

               # Initial inventory must equal given value

subject to Balance {p in PROD, t in 1..T}:
   Make[p,t] + Inv[p,t-1] = Sell[p,t] + Inv[p,t];

               # Tons produced and taken from inventory
               # must equal tons sold and put into inventory
```

From the above picture, {PROD} indicate the param is constricted with the period 1…T and which will be included in the .dat.

```
param T := 4;
set PROD := bands coils;

param avail :=  1 40  2 40  3 32  4 40 ;

param rate :=  bands 200   coils 140 ;
param inv0 :=  bands  10   coils   0 ;

param prodcost :=  bands 10    coils  11 ;
param invcost  :=  bands  2.5  coils   3 ;

param revenue:   1     2     3     4 :=
       bands    25    26    27    27
       coils    30    35    37    39 ;

param market:    1     2     3     4 :=
       bands  6000  6000  4000  6500
       coils  4000  2500  3500  4200 ;
```

T := 4; which shows there are four periods in total

The above picture shows how data are varied regarding the period, which means the cost, revenue, number of production are different in different period.

→ With the use of .dat file, problems with many variables can be easily model under AMPL.

Chapter 3

Sets will be focus in this chapter.

As mentioned above, AMPL is case sensitive, which in all contexts, upper and lower case letters are distinct, i.e. "Fish", "fish", "FISH" are representing three different set members. To declare the set in AMPL, the command is *set.*

e.g
set PROD = {"brands", " coils", "plate"};

"brands", " coils" and "plate" are in the set of PROD.

For numbers, as mentioned above (1…T), {1,2,3,4,5,6} can be also described by 1..6. And for the years, if we want to conclude in the set with every five years from 1990 to 2020, either we use {1990, 1995, 2000, 2005, 2010, 2015, 2020} or 1990…2000 by 5. An additional *by* clause can be used to specify an interval other than 1 between the number.

In such case, we can also indicate in .dat as, i.e,

param start := 1990;
param end := 2020;
param interval :=5;

AMPL has four operators that construct new sets from exisiting ones:

A union B →union: in either A or B
A inter B → intersection: in both A and B
A diff B → difference: in A but not B
A symdiff B →symmetric difference: in A or B but not both

```
ampl: set Y1 = 1990 .. 2020 by 5;
ampl: set Y2 = 2000 .. 2025 by 5;
ampl: display Y1 union Y2, Y1 inter Y2;
set Y1 union Y2 := 1990 1995 2000 2005 2010 2015 2020 2025;
set Y1 inter Y2 := 2000 2005 2010 2015 2020;

ampl: display Y1 diff Y2, Y1 symdiff Y2;
set Y1 diff Y2 := 1990 1995;
set Y1 symdiff Y2 := 1990 1995 2025;
```

AMPL can also define its own ordering by adding the keyword *ordered* or *circular.*

- ***circular*** → the first number is considered to follow the last number;
- ***ordered*** → the first number has not predecessor and the last number has no successor

For example;
{27 sep, 04 oct, 11oct, 18 oct}
can be replaced by
set WEEKS ordered;

Remarks: As the "ordered" suggests, it makes a difference which object comes first; {"Fish", "Egg"} is not the same as {"Egg", "Fish"}

Subsets and slices of ordered pairs

We use {ORIG, DEST} as an example,

set LINKS within {ORIG, DEST};

from this command, it indicates there are different number of links in between ORIG and DEST, see the picture in follows,

	FRA	DET	LAN	WIN	STL	FRE	LAF
GARY		x	x		x		x
CLEV	x	x	x	x	x		x
PITT	x			x	x	x	

The rows represent origins and the columns destinations, which each pair in the set is marked by an x. It shows the links between ORIG and DEST.

```
set ORIG;   # origins
set DEST;   # destinations
set LINKS within {ORIG,DEST};
param supply {ORIG} >= 0;   # amounts available at origins
param demand {DEST} >= 0;   # amounts required at destinations
   check: sum {i in ORIG} supply[i] = sum {j in DEST} demand[j];
param cost {LINKS} >= 0;   # shipment costs per unit
var Trans {LINKS} >= 0;    # units to be shipped
minimize Total_Cost:
   sum {(i,j) in LINKS} cost[i,j] * Trans[i,j];
subject to Supply {i in ORIG}:
   sum {(i,j) in LINKS} Trans[i,j] = supply[i];
subject to Demand {j in DEST}:
   sum {(i,j) in LINKS} Trans[i,j] = demand[j];
```

By adding **set LINKS within {ORIG, DEST};** we can input the data as shown in the following,

```
param: ORIG:  supply :=
    GARY  1400    CLEV  2600    PITT  2900 ;
param: DEST:  demand :=
    FRA    900    DET  1200    LAN    600    WIN  400
    STL  1700    FRE  1100    LAF  1000 ;
param: LINKS:  cost :=
    GARY DET 14   GARY LAN 11   GARY STL 16   GARY LAF  8
    CLEV FRA 27   CLEV DET  9   CLEV LAN 12   CLEV WIN  9
    CLEV STL 26   CLEV LAF 17
    PITT FRA 24   PITT WIN 13   PITT STL 28   PITT FRE 99 ;
```

Comparing the examples shown in chapter 1 and 2, the data of cost is no longer be shown in a matix of ORIG and DEST but altogether with the links.

Chapter 4

In this chapter, parameter, expression and linear program will be shown.

A simple definition of *param* is stated above, which contains numerical value and is called parameter.

param can also be used together with a set, e.g.;
param avail {1..T};
which means there are avail[1], avail[2],…, avail[T]

Types of expression:

Usual style	alternative style	type of operands	type of result
`if-then-else`		logical, arithmetic	arithmetic
`or`	`||`	logical	logical
`exists forall`		logical	logical
`and`	`&&`	logical	logical
`not` (unary)	`!`	logical	logical
`< <= = <> > >=`	`< <= == != > >=`	arithmetic	logical
`in not in`		object, set	logical
`+ - less`		arithmetic	arithmetic
`sum prod min max`		arithmetic	arithmetic
`* / div mod`		arithmetic	arithmetic
`+ -` (unary)		arithmetic	arithmetic
`^`	`**`	arithmetic	arithmetic

Exponentiation and `if-then-else` are right-associative; the other operators are left-associative. The logical operand of `if-then-else` appears after `if`, and the arithmetic operands after `then` and (optionally) `else`.

abs (x)	absolute value, $\lvert x \rvert$
acos (x)	inverse cosine, $\cos^{-1}(x)$
acosh (x)	inverse hyperbolic cosine, $\cosh^{-1}(x)$
asin (x)	inverse sine, $\sin^{-1}(x)$
asinh (x)	inverse hyperbolic sine, $\sinh^{-1}(x)$
atan (x)	inverse tangent, $\tan^{-1}(x)$
atan2 (y, x)	inverse tangent, $\tan^{-1}(y/x)$
atanh (x)	inverse hyperbolic tangent, $\tanh^{-1}(x)$
cos (x)	cosine
cosh (x)	hyperbolic cosine
exp (x)	exponential, e^x
log (x)	natural logarithm, $\log_e(x)$
log10 (x)	common logarithm, $\log_{10}(x)$
max (x, y, ...)	maximum (2 or more arguments)
min (x, y, ...)	minimum (2 or more arguments)
sin (x)	sine
sinh (x)	hyperbolic sine
sqrt (x)	square root
tan (x)	tangent
tanh (x)	hyperbolic tangent

For logical and conditional expressions:

=	equal to
<>	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

Remarks: AMPL uses "." in a data statement to indicate an omitted entry in the table

In linear programs, variables, objectives and constraints are indispensable. However, a set can contain lots of variable. i.e.

var Make {p in PROD} >= 0, <= market [p];

this command means Make is one of the variables in the set PROD, and the parameter of make should be larger than 0 but not exceed the parameter of market.

For objectives, simply consists one of the keywords *minimize* or *maximize*, see Chapter 1 and 2.

And constraints, it begins with the keywords *subject to*, see chapter 1 and 2.

Chapter 5

This chapter is about specifying data in command.

AMPL reads the data statement which are the data input in .dat is initiated by the *data* command. i.e.

Console:
ampl: data diet.dat;

AMPL reads data from a file name diet.dat

- **ONE-DIMENSIONAL SETS AND PARAMETERS, E.G.**

1. parameter

set PROD;
param rate {PROD} > 0;

2. set

set PROD := bands coils plate ;

3. set with parameter

param rate := bands 200 coils 140 plate 160 ;

or

```
param rate :=
        bands  200
        coils    140
        plate   160 ;
```

- **TWO-DIMENSIONAL SETS AND PARAMETERS, E.G.**

1. set

set ORIG; #origins
set DEST; #destination

set ORIG := GARY CLEV PITT ;
set DEST := FRA DET LAN WIN STL FRE LAF ;
GARY, CLEV, PITT are in the set of ORIG; FRA, DET, LAN, WIN, STL, FRE, LAF are in the set of DEST.

2. Parameter with set

```
param cost :=
   GARY DET 14   GARY LAN 11   GARY STL 16   GARY LAF  8
   CLEV FRA 27   CLEV DET  9   CLEV LAN 12   CLEV WIN  9
   CLEV STL 26   CLEV LAF 17   PITT FRA 24   PITT WIN 13
   PITT STL 28   PITT FRE 99 ;
```

e.g. GARY DET 14, which means the cost from GARY to DET is 14 dollar.

- **HIGHER DIMENSIONAL SETS AND PARAMETERS, E.G.**

```
set ROUTES :=
   GARY LAN coils   GARY STL coils   GARY LAF coils
   CLEV FRA bands   CLEV FRA coils   CLEV DET bands
   CLEV DET coils   CLEV LAN bands   CLEV LAN coils
   CLEV WIN coils   CLEV STL bands   CLEV STL coils
   CLEV LAF bands   PITT FRA bands   PITT WIN bands
   PITT STL bands   PITT FRE bands   PITT FRE coils ;
```

For higher dimensional set like the above picture, set should be first defined. Then, we can write the cost corresponding to the set.

e.g.

```
param cost :=
    [*,*,bands]   CLEV FRA 27   CLEV DET  9   CLEV LAN 12
                  CLEV STL 26   CLEV LAF 17   PITT FRA 24
                  PITT WIN 13   PITT STL 28   PITT FRE 99

    [*,*,coils]   GARY LAN 11   GARY STL 16   GARY LAF  8
                  CLEV FRA 23   CLEV DET  8   CLEV LAN 10
                  CLEV WIN  9   CLEV STL 21   PITT FRE 81
```

* represents the ORIG and DEST in this case, however we can also write,

```
param cost :=
    [CLEV,*,bands] FRA 27   DET  9   LAN 12   STL 26   LAF 17
    [PITT,*,bands] FRA 24   WIN 13   STL 28   FRE 99

    [GARY,*,coils] LAN 11   STL 16   LAF  8
    [CLEV,*,coils] FRA 23   DET  8   LAN 10   WIN  9   STL 21
    [PITT,*,coils] FRE 81 ;
```

and in this case, * represents DEST, but also we can write the cost along with the variables,

```
param cost :=
   CLEV DET bands  9   CLEV DET coils  8   CLEV FRA bands 27
   CLEV FRA coils 23   CLEV LAF bands 17   CLEV LAN bands 12
   CLEV LAN coils 10   CLEV STL bands 26   CLEV STL coils 21
   CLEV WIN coils  9   GARY LAF coils  8   GARY LAN coils 11
   GARY STL coils 16   PITT FRA bands 24   PITT FRE bands 99
   PITT FRE coils 81   PITT STL bands 28   PITT WIN bands 13 ;
```

Remarks: As mentioned above, if a dot appears in a table which indicates "no value specified here".

Chapter 6

Commands and interaction with solver will be discussed in this chapter.

As mentioned above, AMPL will only read specified file by using *model* and *data* command along with the file name and type. After that, we can input *display* command to show the required data. To conclude an AMPL session, type *end* or *quit*.

With *let* command, it allows you to change particular data values while leaving the model the same, in the following example, *let* is used to try out the upper bound *f_max["Fish"]* on purchasing of food Fish:

ampl: let f_max["Fish"] := 11; → set upper limit as 11
ampl: solve;
MINOS 5.5: optimal solution found. 1 iterations, objective 73.43818182

ampl: let f_max["CHK"] := 12; →set upper limit as 12
ampl: solve;
MINOS 5.5: optimal solution found. 0 iterations, objective 73.43818182

For removing the input data we are two commands we can use, namely, *delete* and *purge.* For *delete*, it only deletes the listed components while *purge* removes not only the listed components but also all components that depends on them directly, see pictures below

```
ampl: model dietobj.mod;
ampl: data dietobj.dat;
ampl: delete Total_Number, Diet_Min;
```

in this case, Total_Number and Diet_Min will be removed

```
param f_min {FOOD} >= 0;
param f_max {j in FOOD} >= f_min[j];
var Buy {j in FOOD} >= f_min[j], <= f_max[j];
minimize Total_Cost:  sum {j in FOOD} cost[j] * Buy[j];
```

if we type *purge f_min,* the parameter f_min and the components whose declarations refer *f_min*, including parameter *f_max* and variable *Buy*.

Remarks: To check which components depend on some given component, we can type *xref* to find out.

```
ampl: xref f_min;
# 4 entities depend on f_min:
f_max
Buy
Total_Cost
Diet
```

it shows the component *f_min* is depended by *f_max, Buy,Total_Cost* and *Diet* and which when *purge f_min* is applied, the other four components will also be removed.

By using *redeclare* command followed by the complete revised declaration you would like to substitute, it changes the declaration.

Speaking of changing model, there are four *commands, fix, unfix, drop and restore*.

Drop → specify a particular constraint to ignore
Restore → to reverse the effect of *drop*

```
subject to Diet_Max {i in MAXREQ}:
   sum {j in FOOD} amt[i,j] * Buy[j] <= n_max[i];
```

in this case, if we type drop Diet_Max, the whole constraint is ignored.

Fix → fixes specified variables at their current values
Unfix → reverses the effect

```
ampl: let {j in FOOD} Buy[j] := f_min[j];
ampl: fix {j in FOOD: amt["NA",j] > 1200} Buy[j];
ampl: solve;
MINOS 5.5: optimal solution found.
7 iterations, objective 86.92
Objective = Total_Cost['A&P']
```

first, we let the amount of Buy[j] is equal to f_min[j], then we fix the amount of j must larger than 1200.

To display the value, we can simply type *display* follows by component name. E.g.

```
ampl: display Trans;

Trans [*,*]
:          C118 C138 C140 C246 C250 C251 D237 D239 D241 M233 M239 :=
Coullard    1    0    0    0    0    0    0    0    0    0    0
Daskin      0    0    0    0    0    0    0    0    1    0    0
Hazen       0    0    0    1    0    0    0    0    0    0    0
Hopp        0    0    0    0    0    0    1    0    0    0    0
Iravani     0    1    0    0    0    0    0    0    0    0    0
Linetsky    0    0    0    0    1    0    0    0    0    0    0
Mehrotra    0    0    0    0    0    0    0    1    0    0    0
Nelson      0    0    1    0    0    0    0    0    0    0    0
Smilowitz   0    0    0    0    0    0    0    0    0    1    0
Tamhane     0    0    0    0    0    1    0    0    0    0    0
White       0    0    0    0    0    0    0    0    0    0    1
;
```

By typing *display Trans;,* the result of Trans is shown, however, to ignore the zero rows we can type *omit_zero_rows .*
e.g. to ignore the zeros in row 1, we type
ampl: option omit_zero_rows 1;
ampl: display Trans;

```
Trans :=
Coullard  C118   1
Daskin    D241   1
Hazen     C246   1
Hopp      D237   1
Iravani   C138   1
Linetsky  C250   1
Mehrotra  D239   1
Nelson    C140   1
Smilowitz M233   1
Tamhane   C251   1
White     M239   1
;
```

all zeros in row 1 is ignored

Also, there are more options in showing the answer, see below

display_eps	smallest magnitude displayed differently from zero (0)
display_precision	digits of precision to which displayed numbers are rounded; full precision if 0 (6)
display_round	digits left or (if negative) right of decimal place to which displayed numbers are rounded, overriding display_precision ("")
solution_precision	digits of precision to which solution values are rounded; full precision if 0 (0)
solution_round	digits left or (if negative) right of decimal place to which solution values are rounded, overriding solution_precision ("")

By using *display* command, bounds and body can be shown simultaneously. e.g.

```
ampl: display Diet_Min.lb, Diet_Min.body, Diet_Min.ub;
:   Diet_Min.lb Diet_Min.body Diet_Min.ub    :=
A        700       1013.98     Infinity
B1         0        605        Infinity
B2         0        492.416    Infinity
C        700        700        Infinity
CAL    16000      16000        Infinity
;
```

Command *print* and *printf*

print

This command works similar to *display* command, however, *print* allows indexing to be nested within an indexed item. e.g.

```
ampl: print {p in PROD} (p, rate[p], {t in 1..T} Make[p,t]);
bands 200 5990 6000 1400 2000 coils 140 1407 1400 3500 4200
```

the printed products, bands and coils are shown with indexing in the rate as well as the period of Make.

printf

This command is exactly the same as the *print,* except the first print item is a character string that provides formatting instruction. E.g.

```
ampl: print {p in PROD} (p, rate[p], {t in 1..T} Make[p,t]);
bands 200 5990 6000 1400 2000 coils 140 1407 1400 3500 4200
```

Command *show*

By using this command, all components can be shown. E.g.

```
Console
AMPL
ampl: model multmip3.mod;
ampl: show;

parameters:   demand   fcost   limit   maxserve   minload   supply   vcost

sets:   DEST   ORIG   PROD

variables:   Trans   Use

constraints:   Demand   Max_Serve   Min_Ship   Multi   Supply

objective:   Total_Cost

checks: one, called check 1.
ampl:
```

If-then-else statement

This statement conditionally control the execution of statements or groups of statements.

1. If

```
if Make["coils",2] < 1500 then printf "under 1500\n";
```

the statement "under 1500\n" will only be printed out if coils in make is less than 1500.

2. If-else

```
if Make["coils",2] < 1500 then {
   printf "Fewer than 1500 coils in week 2.\n";
   let market["coils",2] := market["coils",2] * 1.1;
}
else
   printf "At least 1500 coils in week 2.\n";
```

if the coils in Make less than 1500, the statement
Fewer than 1500 coils in week 2.\n" will be printed out, however, if the coils is equal or more then 1500, it will print "At least 1500 coils in week 2.\n" .

Loop and terminating a loop

Loop → *for* and *repeat*

Function → *break* and *continue*

1. Continue

It stops the current pass through a for or repeat loop, all further statements in the current pass are skipped, and execution continues with the test that controls the start of the next pass. E.g.

```
if Time[3].dual = previous_dual then continue;
```

the statement only resume if Time[3].dual is equal to previous_dual

2.* Bre*ak

The break statement completely terminates a for or repeat loop, sending control immediately to the statement follow- ing the end of the loop. E.g.

```
if Time[3].dual = 0 then break;
```

the statement stops only if Time[3].dual is equal to 0

Detecting infeasibility in presolve

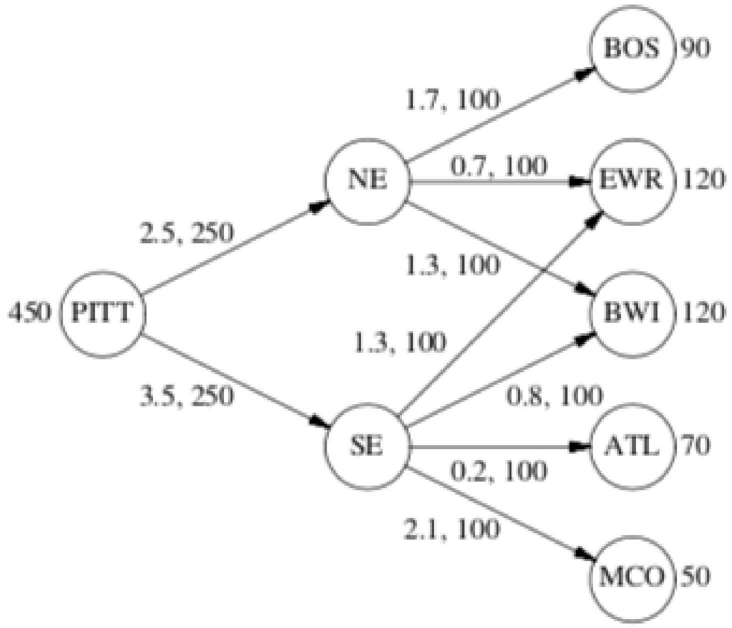
If any variable's lower bound is greater than its upper bound, then there can be no solution satisfying all the bounds and other constraints, and an error message is printed. E.g.

```
ampl: let market["bands"] := 5000;
ampl: let avail := 13;
ampl: solve;
presolve: constraint Time cannot hold:
      body <= 13 cannot be >= 13.2589; difference = -0.258929
```

Chapter 7

This chapter is about Network Linear Programs.

Speaking of network, there are nodes and arrows with directions. E.g.



from the above picture, it shows there is a flow from PITT to BOS via NE, PITT supplies 450 unit of product while BOS demands 90 unit. The capacity from PITT to NE is 250 with the cost 2.5 dollar per unit, while NE to BOS is 100 with the cost 1.7 per unit.

General transshipment model

```
set CITIES;
set LINKS within (CITIES cross CITIES);

param supply {CITIES} >= 0;   # amounts available at cities
param demand {CITIES} >= 0;   # amounts required at cities

check: sum {i in CITIES} supply[i] = sum {j in CITIES} demand[j];

param cost {LINKS} >= 0;      # shipment costs/1000 packages
param capacity {LINKS} >= 0;  # max packages that can be shipped

var Ship {(i,j) in LINKS} >= 0, <= capacity[i,j];
                              # packages to be shipped

minimize Total_Cost:
   sum {(i,j) in LINKS} cost[i,j] * Ship[i,j];

subject to Balance {k in CITIES}:
   supply[k] + sum {(i,k) in LINKS} Ship[i,k]
      = demand[k] + sum {(k,j) in LINKS} Ship[k,j];
```

Specialized transport model

```
set D_CITY;
set W_CITY;
set DW_LINKS within (D_CITY cross W_CITY);

param p_supply >= 0;            # amount available at plant
param w_demand {W_CITY} >= 0;   # amounts required at warehouses

   check: p_supply = sum {j in W_CITY} w_demand[j];

param pd_cost {D_CITY} >= 0;    # shipment costs/1000 packages
param dw_cost {DW_LINKS} >= 0;

param pd_cap {D_CITY} >= 0;     # max packages that can be shipped
param dw_cap {DW_LINKS} >= 0;

var PD_Ship {i in D_CITY} >= 0, <= pd_cap[i];
var DW_Ship {(i,j) in DW_LINKS} >= 0, <= dw_cap[i,j];
                                # packages to be shipped

minimize Total_Cost:
   sum {i in D_CITY} pd_cost[i] * PD_Ship[i] +
   sum {(i,j) in DW_LINKS} dw_cost[i,j] * DW_Ship[i,j];

subject to P_Bal:  sum {i in D_CITY} PD_Ship[i] = p_supply;

subject to D_Bal {i in D_CITY}:
   PD_Ship[i] = sum {(i,j) in DW_LINKS} DW_Ship[i,j];

subject to W_Bal {j in W_CITY}:
   sum {(i,j) in DW_LINKS} DW_Ship[i,j] = w_demand[j];
```

Comparing two models, we simply model the problem of shipments from city to city and the set of cities as well as a set of links in general model. While, the pattern of supplies, demands and the links between cities are accommodated in specialized model.

Example data in 2 models

General:

```
set CITIES := PITT  NE SE  BOS EWR BWI ATL MCO ;

set LINKS := (PITT,NE) (PITT,SE)
             (NE,BOS) (NE,EWR) (NE,BWI)
             (SE,EWR) (SE,BWI) (SE,ATL) (SE,MCO);

param supply  default 0 := PITT 450 ;

param demand  default 0 :=
  BOS  90,  EWR 120,  BWI 120,  ATL  70,  MCO  50;

param:      cost  capacity  :=
  PITT NE    2.5    250
  PITT SE    3.5    250

  NE  BOS    1.7    100
  NE  EWR    0.7    100
  NE  BWI    1.3    100

  SE  EWR    1.3    100
  SE  BWI    0.8    100
  SE  ATL    0.2    100
  SE  MCO    2.1    100  ;
```

Specialized:

```
set D_CITY := NE SE ;

set W_CITY := BOS EWR BWI ATL MCO ;

set DW_LINKS := (NE,BOS) (NE,EWR) (NE,BWI)
                (SE,EWR) (SE,BWI) (SE,ATL) (SE,MCO);

param p_supply := 450 ;

param w_demand :=
  BOS  90,  EWR 120,  BWI 120,  ATL  70,  MCO  50;

param:  pd_cost  pd_cap :=
  NE       2.5     250
  SE       3.5     250 ;

param:     dw_cost  dw_cap :=
  NE BOS     1.7     100
  NE EWR     0.7     100
  NE BWI     1.3     100

  SE EWR     1.3     100
  SE BWI     0.8     100
  SE ATL     0.2     100
  SE MCO     2.1     100 ;
```

From the general data, only the linkage with demand and supply is shown, however, it does not show the warehouse and plants. As for the specialized case, D_CITY and W_CITY indicate distribution center and warehouse which is shown clearly the direction of the transportation flow.

Moreover, we can also use *node* and *arc* to represent the set of cities and the link repectively. In the following examples, it shows the *node* and *arc* in two models

1. Gerneral

```
set CITIES;
set LINKS within (CITIES cross CITIES);

param supply {CITIES} >= 0;   # amounts available at cities
param demand {CITIES} >= 0;   # amounts required at cities

   check: sum {i in CITIES} supply[i] = sum {j in CITIES} demand[j];

param cost {LINKS} >= 0;      # shipment costs/1000 packages
param capacity {LINKS} >= 0;  # max packages that can be shipped

minimize Total_Cost;

node Balance {k in CITIES}: net_in = demand[k] - supply[k];

arc Ship {(i,j) in LINKS} >= 0, <= capacity[i,j],
   from Balance[i], to Balance[j], obj Total_Cost cost[i,j];
```

2. Specialised

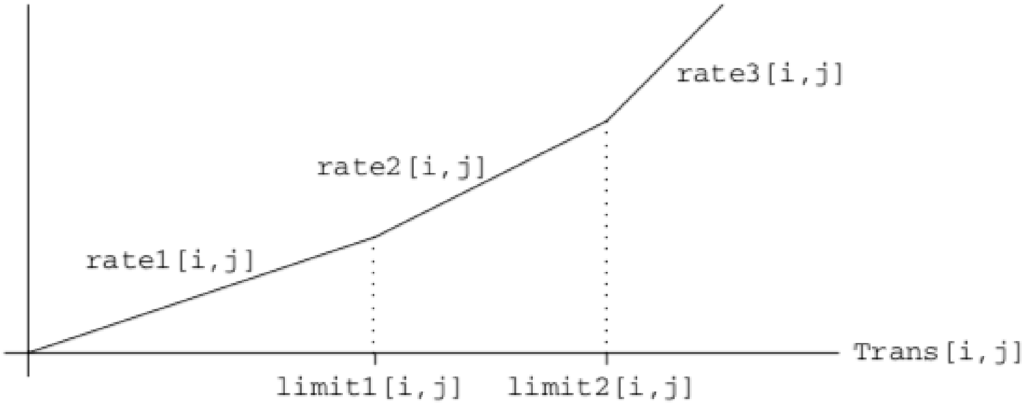
```
set D_CITY;
set W_CITY;
set DW_LINKS within (D_CITY cross W_CITY);

param p_supply >= 0;            # amount available at plant
param w_demand {W_CITY} >= 0;   # amounts required at warehouses

   check: p_supply = sum {j in W_CITY} w_demand[j];

param pd_cost {D_CITY} >= 0;    # shipment costs/1000 packages
param dw_cost {DW_LINKS} >= 0;

param pd_cap {D_CITY} >= 0;     # max packages that can be shipped
param dw_cap {DW_LINKS} >= 0;

minimize Total_Cost;

node Plant: net_out = p_supply;

node Dist {i in D_CITY};

node Whse {j in W_CITY}: net_in = w_demand[j];

arc PD_Ship {i in D_CITY} >= 0, <= pd_cap[i],
   from Plant, to Dist[i], obj Total_Cost pd_cost[i];

arc DW_Ship {(i,j) in DW_LINKS} >= 0, <= dw_cap[i,j],
   from Dist[i], to Whse[j], obj Total_Cost dw_cost[i,j];
```

From using *node* in this case, we can find out the net input of every node.

Chapter 8

This chapter is about Piecewise-Linear Programs

We use transportation cost as an example, in reality, it is impossible to have a constant cost. To express the different costs graphically, it is similar to the picture in follows,



This is a cumulative graph with axis-x represent the number of transport units while axis-y is the cost. The slopes shows the difference of the cost per unit with the limit 1 and 2.

To express it in AMPL,

```
set ORIG;    # origins
set DEST;    # destinations

param supply {ORIG} >= 0;   # amounts available at origins
param demand {DEST} >= 0;   # amounts required at destinations

   check: sum {i in ORIG} supply[i] = sum {j in DEST} demand[j];

param rate1 {i in ORIG, j in DEST} >= 0;
param rate2 {i in ORIG, j in DEST} >= rate1[i,j];
param rate3 {i in ORIG, j in DEST} >= rate2[i,j];

param limit1 {i in ORIG, j in DEST} > 0;
param limit2 {i in ORIG, j in DEST} > limit1[i,j];

var Trans {ORIG,DEST} >= 0;    # units to be shipped

minimize Total_Cost:
   sum {i in ORIG, j in DEST}
      <<limit1[i,j], limit2[i,j];
        rate1[i,j], rate2[i,j], rate3[i,j]>> Trans[i,j];

subject to Supply {i in ORIG}:
   sum {j in DEST} Trans[i,j] = supply[i];

subject to Demand {j in DEST}:
   sum {i in ORIG} Trans[i,j] = demand[j];
```

Remark: Other function of AMPL applying in Piecewise-linear program is the same.

Chapter 9

This chapter is about Nonlinear programs
Comparing with linear program, the degree of variable is 2 or higher in nonlinear program.

```
var Cost {ORIG,DEST};          # shipment costs per unit
var Ship {ORIG,DEST} >= 0;     # units to ship

minimize Total_Cost:
   sum {i in ORIG, j in DEST} Cost[i,j] * Ship[i,j];
```

For example, the Total cost from above picture shows that it is the sum of different cost times corresponding ship.

```
subject to Cost_Relation {i in ORIG, j in DEST}:
   Cost[i,j] =
      (cost1[i,j] + cost2[i,j]*Ship[i,j]) / (1 + Ship[i,j]);
```

However in such case, this is not longer a linear objective, as there are 2 different variables representing cost.

→ The approach to formulate non-linear program in AMPL is similar in mathematical way. See below,

```
minimize Total_Cost:
   sum {(i,j) in LINKS} cost[i,j] * Ship[i,j] +
   sum {k in CITIES} pen * Discrepancy[k] ^ 2;
```

Yet, if we are formulating inherently linear program, we should be careful in inputting the formulas in AMPL. See the following example

As a simple example, a model of a natural gas pipeline network must incorporate not only the shipments between cities but also the pressures at individual cities, which are subject to certain bounds. Thus in addition to the flow variables Ship[i,j] the model must define a variable Press[k] to represent the pressure at each city k. If the pressure is greater at city i than at city j, then the flow is from i to j and is related to the pressure by

```
Flow[i,j]^2 = c[i,j]^2 * (Press[i]^2 - Press[j]^2)
```

Remarks: In such cases, using let, display, option etc commands which have mentioned above would be helpful in showing desired data.

Chapter 10

This chapter is about solver PATH and CPLEX

PATH → is for complementarity problem

```
set PROD;   # products
set ACT;    # activities

param cost {ACT} > 0;       # cost per unit of each activity
param demand {PROD} >= 0;  # units of demand for each product
param io {PROD,ACT} >= 0;  # units of each product from
                           # 1 unit of each activity

param level_min {ACT} > 0; # min allowed level for each activity
param level_max {ACT} > 0; # max allowed level for each activity

var Price {i in PROD};
var Level {j in ACT};

subject to Pri_Compl {i in PROD}:
   Price[i] >= 0 complements
      sum {j in ACT} io[i,j] * Level[j] >= demand[i];

subject to Lev_Compl {j in ACT}:
   level_min[j] <= Level[j] <= level_max[j] complements
      cost[j] - sum {i in PROD} Price[i] * io[i,j];
```

The above picture shows the complementarity problem which the level is bounded and also has to fulfil the constraints of minimum level and maximum level.

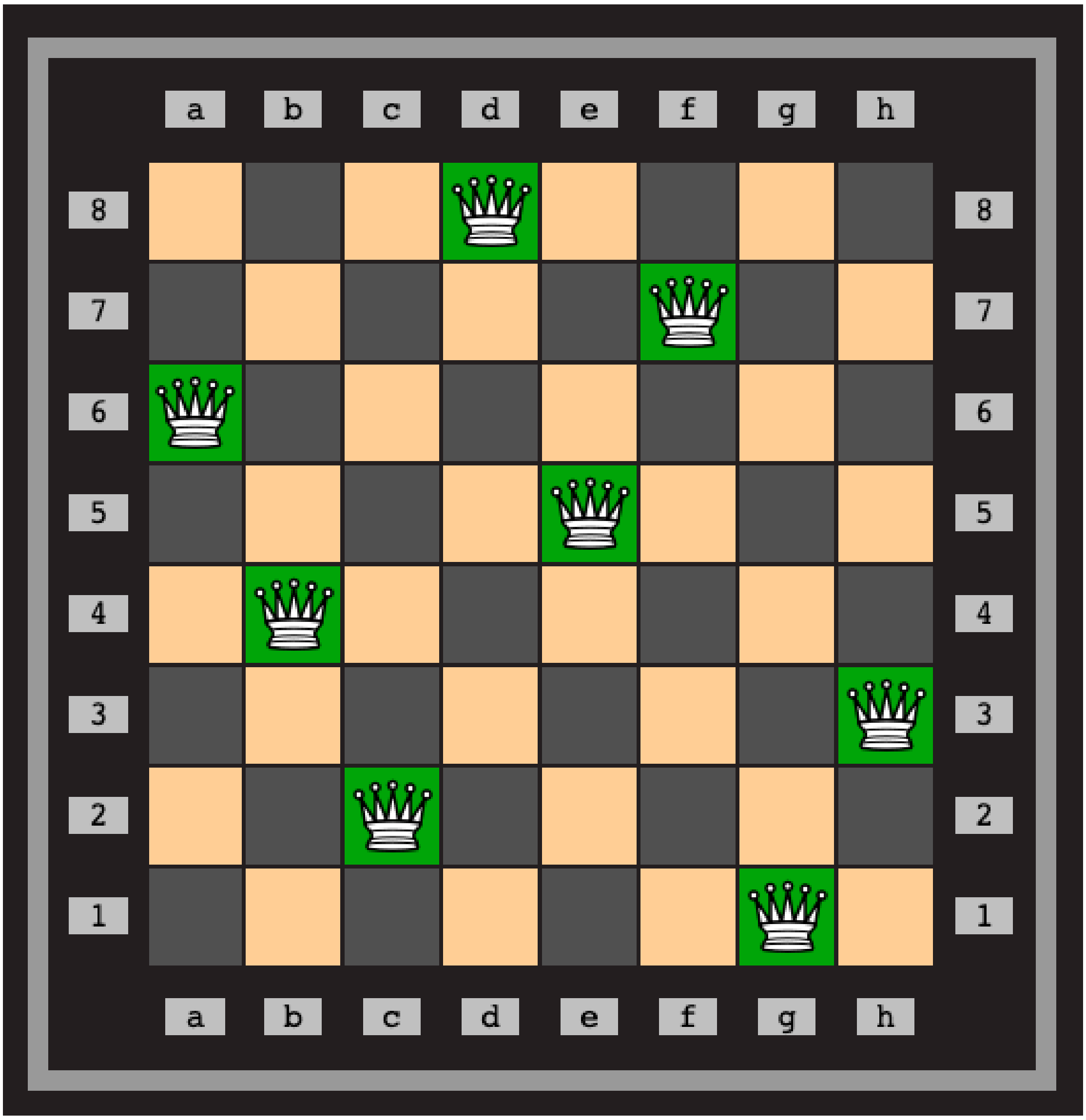
Applying the solver **PATH**, the complementarity problem can be solved as the same solution as above mentioned. For example;

```
ampl: model econ2.mod;
ampl: data econ2.dat;
ampl: option solver path;
ampl: solve;
Path v4.5: Solution found.
9 iterations (4 for crash); 8 pivots.
22 function, 10 gradient evaluations.

ampl: display level_min, Level, level_max;
:    level_min   Level  level_max   :=
P1        240      240      1000
P1a       270     1000      1000
P2        220      220      1000
P2a       260      680      1000
P2b       200      200      1000
P3        260      260      1000
P3c       220     1000      1000
P4        240      240      1000
;
```

CPLEX → is for integer programming problem

For example, solving eight queen problem with AMPL

Using a regular chess board, the challenge is to place eight queens on the board such that no queen is attacking any of the others. (For those not familiar with chess pieces, the queen is able to attack any square on the same row, any square on the same column, and also any square on either of the diagonals).



```
# eightqueens.mod
param n >= 0, default 8;
set N := 1..n;
var x{N,N} binary;
maximize queens : sum{i in N, j in N} x[i,j];
subject to rows {i in N} : sum{j in N} x[i,j] <= 1;
subject to cols {j in N} : sum{i in N} x[i,j] <= 1;
subject to diagNW {i in N, j in N} :
  sum{h in N : h < i and h < j} x[i-h,j-h] +
    sum{h in N : h+i<=n and h+j<=n} x[i+h,j+h] <= 1;
subject to diagSW {i in N, j in N} :
  sum{h in N : h < i and h+j<=n} x[i-h,j+h] +
    sum{h in N : h+i<=n and h < j} x[i+h,j-h] <= 1;
```

See below, solving eight queen problem without using CPLEX solver,

```
ampl: model eightq.mod;
ampl: solve;
MINOS 5.51: ignoring integrality of 64 variables
MINOS 5.51: optimal solution found.
51 iterations, objective 8
ampl: display queens;
queens = 8

ampl: display x;
x [*,*]
:     1          2             3            4          5          6       :=
1   0.578125   -8.85387e-18   0            0.421875   0          0
2   0           0.15625       0            0          0.429687   0.164063
3   0.109375    0             4.30633e-17  0          0.570313   0.0625
4   0           0.273438      0            0.265625   0          0
5   0           0             0            0          0          0.484375
6   0.3125      0             0.6875       0          0         -1.09814e-16
7   0          -1.11022e-16   0.171875     0.3125     0          0
8   0           0.570312      0.140625     0          0          0.289063

:      7              8          :=
1   -1.37281e-16    0
2    0.25           0
3    0.257813      -1.37281e-16
4    0.0625         0.398437
5    0.429687       0.0859375
6    0              0
7    0              0.515625
8    0              0
;
```

CPLEX solver provide an integer solution.

For example in this case, 1 and 0 will be used to represent the exist of the chess. Using the chess board picture as an example,

	1	2	3	4	5	6	7	8
1	0	0	0	1	0	0	0	0
2	0	0	0	0	0	1	0	0
3	1	0	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0
5	0	1	0	0	0	0	0	0
6	0	0	0	0	0	0	0	1
7	0	0	1	0	0	0	0	0
8	0	0	0	0	0	0	1	0

Remark: As same as using solver PATH, while using CPLEX, command has to be stated, i.e. *option solver cplex;*

Bibilography

Robert Fourer, David M. Gay,Brian W. Kernighan, *AMPL A Modeling Language for Mathematical Programming*, Duxbury Press / Brooks/Cole Publishing Company, 2002.

Leo Liberti, *Problems and exercises in Operations Research,* 2006.

Data Genetics, Eight queens problem, Retrieved from
http://www.datagenetics.com/blog/august42012/